CS 421 Lecture 2: More OCaml

- Announcements

- Lecture outline
  - Types
  - let expressions
  - Scoping rules
  - Tuples and pattern-matching
  - Lists and pattern-matching

Announcements

- Reminder: MP1 due 1:00PM CDT Wednesday
  - EWS machines to use: remlnx, gllnx1-40 (.ews.uiuc.edu)
- No “live” lectures next Monday & Tuesday (June 8, 9)
  - Pre-recorded lecture videos will be posted on the web site
- Limited course staff availability this weekend
  - Friday – Sunday you are on your own!

More OCaml

- Functional language – rely on *expression evaluation* rather than *statement execution*
  - Heavy use of recursion
  - Type inference
  - Dynamic memory allocation
  - “Higher-order functions” (will be covered in the second half of the course)

Types

- Basic: int, string, …
- Function: $\tau_1 \rightarrow \tau_2 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau$
  - *e.g.* int → int → int
- Later in this class: tuples, lists

Let expressions

- At "top level," use let to define variables and functions
- Use "let rec" for recursive definitions, *e.g.*:

```
let rec sumsqrs m =
    if m=0 then 0 else m*m + sumsqrs (m-1);;
```

Nested let definitions

```
let f x y = let z = sqrt(x+y)
            in x*z;;

let f x y = let f' a = a ^ "\n"
            in f' (x^y)

let sumsqrs n =
   let rec aux m =
      if m>n then 0
      else m*m + aux (m+1)
   in aux 1;;
```

Further Examples

```
let z = …
and t = …
in … z … t …

let f x =
   let f' y = …
   in let f'' z = …
      in … f' … f'' …
```

Scope

- Set of variables accessible at a given point.
- Let's look at Java first. Basic rule: *closest enclosing declaration.*

```
class A {
    int x=3;
    void foo(int x) {
        System.out.println(x);
        for(int i=0; i<5; i++) {
            System.out.println(i);
        }
        System.out.println(i);
    }
}
```

Scope in OCaml

- Basic rule is the same, *e.g.*:

```
let x = 5;;
let f x = let x = 7
          in print_int x;;
```

Scoping rules in OCaml

- Top level:

`let x = … ;;` Scope of x?

`let f a = … ;;` Scope of f, a?

Scope of e

`e : let x =` e_1 `in` e_2

Scope of e + x

e – all names defined up to this point

Scoping rules in OCaml

Scope of ??

e : `let f x =` e_1 `in` e_2

Scope of ??

Scope of ??

e : `let rec f x =` e_1 `in` e_2

Scope of ??

Why let rec?

- To understand let rec, consider this definition:

```
let f x = x*x;;
let f x = … f (x-1) … in …
```

- It is legal if the entire let expression is in the scope of a definition of `f` (with the right type).
- In that case, the expression `f(x-1)` refers to the *prior* definition of `f` – not what we intended!

Mutual recursion

- Does this work?

```
let rec even n =
   if n=0 then true
   else odd (n-1)
and odd n =
   if n=0 then false
   else even(n-1);;
```

- And this?

```
let rec even n =
   let rec odd n =
      if n=0 then …
   in if n=0 then …
```

- What's different here… vs. here?

Tuples in OCaml

- Consider structs in C, or Java classes with public fields and no methods (and just one constructor).
- Example:

```
class Pr { public int x;
           public string x;
           public Pr(int x, int s) {
              this.x = x; this.s = s;
           }
         }
```

- Purpose: put several values together into a single object that can be passed to, or returned from, methods.

Tuples

- In Java, clients of class Pr do this:

```
Pr p = new Pr(3, “tim”);
… p.x … p.s …
```

- Ocaml: create pair with no calss definition needed:

```
let p = (3,“tim”)
… fst p … snd p …
```

- Type of p is “int * string”
- Pairs in Ocaml serve same purpose as structs in C, Java

Tuples

- Can have as many values as you wish in a tuple:

```
(3,"rick", 4.0) : int * string * float

("ted", "bill") : string * string

let b = (3,('a',4)) : ??
```

- How would we extract 'a' from this?

- However, functions fst and snd work *only* on pairs. To define functions on other tuples you need…

Pattern matching

- Two ways to define the same function

```
let sum p = (fst p) + (snd p)
let sum (a,b) = a+b
```

  - Both define the same function of type int * int → int

- Examples:

```
let fst_of_3 (x,y,z) = x;;



let incr_fst_of_3 (x,y,z) = x+1;;
```

“Polymorphic” types

```
let fst_of_3 (x,y,z) = x;;

    ‘a * ‘b * ‘c → ‘a


let incr_fst_of_3 (x,y,z) = x+1;;

    int * ‘a * ‘b → int
```

Curried vs. Uncurried functions

```
let f x y = … x … y …
```

curried form

```
let g(x,y) = … x … y …
```

uncurried form

- Wrong usage:

```
f (1,2)
g 1 2
```

“match” expressions

- Another way to use pattern-matching to define functions:

```
let fst_of_3 x =
   match x with
      (a,b,c) -> a;;
```

- But match expressions allow *alternates*:

```
let rec fib n =
   match n with 0 -> 1
              | 1 -> 1
              | _ -> fib(n-2)+fib(n-1);;
```

Lists

- Linked lists in Java:

```
class List {
   int head;
   List tail;
   static List cons(int x, List y) {
      List lst = new List();
      lst.head = x;
      lst.tail = y;
      return lst;
   }
}
```

```
List lst1 = List.cons(3,null);
lst1.head = 3;
List lst2 = List.cons(4,lst1);
List lst3 = List.cons(5,lst2);
```

Recursive functions in Java

```
List lst1 = List.cons(3,null);
lst1.head = 3;
List lst2 = List.cons(4,lst1);

int sum(List L) {
   if (L==null)
   then return 0;
   else return L.head + sum(L.tail);
}
```

or…

```
int sum(List L) {
   return L==null ? 0 : L.head + sum(L.tail);
}
```

Recursive functions in Java

- Exercise: define Append(List x, List y)

```
List Append(List x, List y) {







}
```

Lists in OCaml

- Built-in data type
- Syntax
  - [] – empty list
  - [a; b; …; c] – list with elements a, b, …, c
  - a :: x – list obtained by putting a on the front of list x (“consing”)

- Examples

```
let lst1 = [];;
let lst2 = [3];;
let lst1 = lst2;;
let lst3 = 5::4::lst2;;
lst3 = [5;4;3];;
```

Pattern-matching on lists

```
let f[a;b] = …
let g(x::xs) = …
let h(x::y::xs) = …
let f x = match x with [] -> …
                     | y::ys -> …
```

- Example:

```
let rec sum x =
   match x with [] -> 0
              | y::ys -> y + sum ys;;
```

Append

```
let rec append x y =
     match x with [] -> y
                | z::zs -> z :: (append zs y);;
```

- Compare the Ocaml functions to the Java functions…

Tuples vs. lists

- Tuples are fixed-size, heterogenous collections

- Lists are extendable, homogeneous collections